A Comparison of Scheduling Approaches for Mixed-Parallel Applications on Heterogeneous Platforms

Tchimou N'Takpé, Frédéric Suter, Henri Casanova

A Comparison of Scheduling Approaches for Mixed-Parallel Applications on Heterogeneous Platforms

Tchimou N'Takpé [*] Frédéric Suter
Nancy University / LORIA
UMR 7503 CNRS - INPL - INRIA - Nancy 2 - UHP, Nancy 1
[Frederic.Suter,Tchimou.Ntakpe]@loria.fr

Henri Casanova[†]
Dept. of Information and Computer Sciences
University of Hawai'i at Manoa
henric@hawaii.edu

Abstract

Mixed-parallel applications can take advantage of large-scale computing platforms but scheduling them efficiently on such platforms is challenging. In this paper we compare the two main proposed approaches for solving this scheduling problem on a heterogeneous set of homogeneous clusters. We first modify previously proposed algorithms for both approaches and show that our modifications lead to significant improvements. We then perform a comparison of the modified algorithms in simulation over a wide range of application and platform conditions. We find that although both approaches have advantages, one of them is most likely the most appropriate for the majority of users.

1. Introduction

The use of parallel computing for large and time-consuming scientific simulations has become mainstream. Two kinds of parallelism are typically exploited in scientific applications: *task parallelism* and *data parallelism*. In task parallelism the application is partitioned into a set of tasks organized in a Directed Acyclic Graph (DAG) in which edges correspond to precedence and/or data communication constraints. In data parallelism an application exhibits parallelism typically at the level of loops, meaning that loop iterations can be executed, at least conceptually, in a Single Instruction Multiple Data (SIMD) fashion. In practice each kind of parallelism corresponds to a specific programming model. A way to expose increased parallelism, to in turn achieve higher scalability and performance, is to write parallel applications that use both task and data parallelism. This approach is termed *mixed parallelism* and allows several data-parallel tasks to be executed concurrently. Mixed parallelism arises naturally in many applications and we refer the reader to [3] for application examples and a quantitative discussion of the benefits of mixed parallelism.

A well-known challenge for the efficient execution of task-parallel applications is scheduling. The problem consists in deciding which compute resource should perform which task when, in a view to optimizing some metric such as overall execution time. In the case of mixed-parallel applications, data parallelism adds a level of difficulty to the task-parallel scheduling problem. Indeed, the common case is that data-parallel tasks are moldable, i.e., they can be executed on arbitrary numbers of processors, with more processors leading to faster task execution times. This raises the question: how many processors should be allocated to each data-parallel task? There is thus an intriguing tension between running more concurrent data-parallel tasks with each fewer processors, or fewer concurrent data-parallel tasks with each more processors. Not surprisingly this scheduling problem is NP-complete (2-optimal algorithms are known) [7, 8, 14]. Consequently, several researchers have attempted to design scheduling heuristics for mixed-parallel applications. The most successful approaches proceed in two phases: one phase to determine how many processors should be allocated to each data-parallel task, another phase to schedule these tasks on the platform using standard list scheduling algorithms.

A limitation of these two-phase scheduling algorithms is that they assume a homogeneous computing environment. While homogeneous platforms are relevant to many real-world scenarios, in the face of increasing computation and

[*] Supported in part by the Conseil Régional de Lorraine and the Government of Côte d'Ivoire.

[†] Supported in part by the NSF under Award 0546688.

memory demands of scientific application, many current computing platforms consist of multiple compute clusters aggregated within or across institutions. Mixed parallel applications appear then ideally positioned to take advantage of such large-scale platforms. However, the clusters in these platforms are rarely identical (e.g., there can be large slow clusters and small fast clusters).

Two approaches have been recently proposed to schedule mixed-parallel applications on heterogeneous platforms. The first approach consists in adapting the aforementioned two-phase algorithms for mixed-parallel applications on homogeneous platforms and making them amenable to heterogeneous platforms [9]. The second approach consists in adapting list scheduling algorithms that were specifically designed for executing task-parallel applications on heterogeneous platforms and making them amenable to mixed parallelism [2]. Both approaches have merit and the question we answer in this paper is: is one approach significantly better than the other, and if so, which one? We also identify limitations of these algorithms and make several improvements. More specifically, our contributions are:

- We improve the M-HEFT algorithm [2] by proposing and evaluating three ways in which the number of processors allocated to a data-parallel task can be reduced in a sensible manner.
- We improve the HCPA two-phase algorithm [9] in two ways. First we implement a more efficient stopping criterion for the first phase. Second, we modify the second phase so that it attempts to use fewer processors than computed in the first phase.
- We perform empirical comparisons of these improved versions of M-HEFT and HCPA via extensive simulations for many application and platform scenarios.

This paper is organized as follows. Section 2 defines our models and our scheduling problem. Section 3 reviews related work and Section 4 describes our improvements to M-HEFT and HCPA. Section 5 presents our experimental results. Finally, Section 6 summarizes our findings and gives perspectives on future work.

2. Problem Statement

We consider a computing platform that consists of c clusters, where cluster $C_k, k = 1, \ldots, c$ contains p_k identical processors. A processor in cluster C_k computes at speed r_k, which is defined as the ratio between that processor's computing speed (in operations per seconds) to that of the slowest processor over all c clusters, which we call the reference processor speed. Clusters may be built with different interconnect technologies and are interconnected together via a high-capacity backbone. Each cluster is connected to the backbone by a single network link. Inter-cluster communications happen concurrently, possibly causing contention on these network links.

A mixed-parallel application is modeled as a DAG $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, where $\mathcal{N} = \{t_i \,|\, i = 1, \ldots, N\}$ is a set of nodes representing data-parallel tasks, or "tasks" for short, and $\mathcal{E} = \{e_{i,j} \,|\, (i,j) \in \{1, \ldots, N\} \times \{1, \ldots, N\}\}$ is a set of edges between nodes, representing communication between tasks. Each edge $e_{i,j}$ has a weight, which is the amount of data (in bytes) that task t_i must send to task t_j (we call t_j a *successor* of t_i and t_i a *predecessor* of t_j). Note that in addition to data communication itself, there may be an overhead for data redistribution, e.g., when task t_i is executed on a different number of processors than task t_j. Without loss of generality we assume that the DAG has a single entry task and a single exit task. Since data-parallel tasks can be executed on various numbers of processors, we denote by $T^k(t, n)$ the execution time of task t if it were to be executed on n processors of cluster C_k. $T^k(t, n)$ accounts for both the computation and the communication costs involved when executing task t on cluster C_k. In practice, $T^k(t, n)$ can be measured via benchmarking on each cluster for several values of n and/or can be calculated via a performance model. The overall execution time is defined as the time between the beginning of the application's entry task and the completion of the application's exit task.

All previous work on mixed scheduling for heterogeneous platforms assumes that a data-parallel task must be executed within a single (homogeneous) cluster. We also make this assumption, which is reasonable because the latency of inter-cluster communications has a high impact on the performance of most data-parallel tasks and an intra-cluster execution is largely preferable.

Given a platform and an application we define the mixed parallelism scheduling problem as follows: for each task, determine the time at which it should start on which cluster and with how many processors, so that data dependencies are respected and the overall execution time is minimized. We assume *space sharing without preemption*, meaning that a processor can be used for at most one task at a time, and that once a task has started execution on a processor it runs to completion. As a result, some processors in a cluster may be idle because tasks that are ready to be executed do not "fit" and must wait for other tasks to complete. $N^k(t)$ denotes the number of processors allocated to task t scheduled on cluster C_k.

3. Previous Work

While a few authors have studied the scheduling of mixed-parallel applications from a theoretical perspective [7, 8, 14], several practical scheduling algorithms have been described in the literature [10, 11, 12]. Most of these

algorithms proceed in two phases. In the first phase the algorithm computes the optimal number of processors for each data-parallel task of the application. In the second phase, the tasks are scheduled using one of the popular list scheduling algorithms. Note that a one-phase algorithm is proposed in [1]. Previously published results show that the CPA algorithm described in [10] leads to the best schedules.

All the algorithms above assume a homogeneous platform, and thus are not applicable to our scheduling problem. Two algorithms have been recently proposed to schedule mixed-parallel applications on heterogeneous platforms: HCPA [9] and M-HEFT [2]. Since we improve on and compare both algorithms, we describe each in detail hereafter.

3.1. The HCPA Algorithm

The HCPA algorithm is an adaptation of the CPA algorithm. We first give a short overview of CPA before describing the salient features of HCPA. (Full details on both algorithms can be found in [10] and [9].) CPA attempts to achieve the best tradeoff between the length of the critical path, T_{CP}, and the average processor utilization, T_A. The bottom-level of a task t, $T_b(t)$ is the length of the path from the task to the exit node, that is the sum of execution times of the tasks along this path. T_{CP} is the maximum bottom level over all tasks in the DAG. T_A is defined as a sum of terms, where each term is the product of a task's execution time by the number of processors allocated to it, for all tasks, divided by the total number of processors. More formally, using the notations defined in Section 2 without the k indices since CPA considers a single cluster:

$$T_{CP} = \max_{t \in \mathcal{N}} T_b(t), \text{ and } T_A = \frac{1}{p} \sum_{t \in \mathcal{N}} T(t, N(t)) \times N(t).$$

The main intuition behind CPA is that both T_{CP} and T_A are lower bounds on the overall execution time, implying that $\max\{T_{CP}, T_A\}$ should be minimized. Recall that CPA proceeds in two phases, where the first phase determines $N(t)$ for each task. Given that T_{CP} decreases and T_A increases as more processors are allocated to tasks, the allocation phase in CPA proceeds as follows. Initially, one processor is allocated to each task. Then CPA identifies which task on the critical path would benefit the most from one extra processor, i.e., for which t would $T(t, N(t))/N(t)$ decrease the most if $N(t)$ were to be increased by one. The allocation of that task is increased by one and this process is repeated while $T_{CP} \geq T_A$. The intuition here is that the best execution time is achieved when both its lower bounds are equal. During the second phase, tasks are scheduled in decreasing order of their bottom levels, which is a classical list scheduling approach [13]. Note that in this phase it is now possible, and in fact necessary, to account for data communication and redistribution costs to make judicious task mapping decisions. In what follows we describe how both phases are adapted to heterogeneous platforms in the HCPA algorithm.

First Phase of HCPA – A difficulty with a heterogeneous platform is that processor allocations can be on clusters with different processor speeds. To remove this difficulty, HCPA reasons about allocations using an equivalent "reference" homogeneous cluster. This reference cluster consists of processors with reference speed, that is the speed of the slowest processor in the original platform. The reference cluster contains more processors than the total number of processors in the heterogeneous platforms. HCPA just uses the CPA processor allocation phase on the reference cluster. Then, processor allocations on the reference cluster are "translated" to allocations on the original clusters. Such a translation is only possible with an analytical model for $T(t, n)$ as a function of n for all task t. The authors in [9] use a popular model for the execution time of many data-parallel applications based on Amdahl's law, meaning that a fraction α of the sequential task execution time cannot be parallelized. A direct application of Amdahl's law shows that, given an allocation on the reference cluster, one can compute an equivalent allocation on any cluster C_k, i.e., an allocation that leads to the same task execution time.

It is important to note that an allocation on the reference cluster may end up being so large that it cannot be translated into any feasible allocation on any cluster C_k. For this reason, HCPA stops increasing allocations on the reference cluster when no cluster C_k could accommodate a translation of the reference allocation. This, in essence, provides another stopping criterion for the allocation phase (i.e., when all allocations are made as large as possible before the $T_{CP} < T_A$ condition occurs).

Second Phase of HCPA – The second phase of the HCPA algorithm follows the same principle as that of CPA: tasks are considered in order of decreasing bottom-level using the reference allocations. However, once a task is selected for scheduling, all its feasible allocations on the clusters of the heterogeneous platforms are determined using the translation procedure described earlier. The task is then scheduled on the cluster that minimizes its completion time.

3.2. The M-HEFT Algorithm

The M-HEFT algorithm takes a simpler approach than HCPA It uses the HEFT [13] algorithm as a starting point. HEFT is a list scheduling algorithm for scheduling a DAG of sequential tasks onto a heterogeneous set of processors. Recall that the bottom-level of a task is the length of the longest path from that task to the exit node. In the case of HEFT the length of a path is defined as the sum of the average computation time of each task and the average communication time of each communication edge along the path,

where averages are computed over all processors and network links. Tasks are scheduled in order of decreasing bottom-levels. Each task is scheduled using the allocation that minimizes its completion time, accounting for time spent in communication.

M-HEFT extends HEFT to the case of data-parallel tasks and a platform that consists of heterogeneous clusters as follows. Instead of computing average task execution times over available processors, M-HEFT simply computes averages over all possible 1-processor allocations over all clusters. Average communication times are computed over the set of communication times between all possible 1-processor allocations (not accounting for data redistribution costs). Each task is then scheduled on the set of processors that minimizes its completion time, accounting for the costs of data redistribution and communication.

4. Algorithm Improvements

4.1. Improved HCPA

Improving the first phase – Recall from Section 3 that the first phase of HCPA computes T_A, which is used in the first phase's stopping criterion. The value of T_A is the ratio of the data-parallel task "areas" to the total number of processors, all for the reference cluster. In practice we found that when the total number of processors in the reference cluster is significantly larger than the number of application tasks, the first phase does not stop soon enough. The obtained schedules are then inefficient (i.e., many extra resources are used that do not contribute to reducing execution time significantly). We address this concern as follows. Instead of taking the ratio to the number of processors in the reference cluster, p_{ref}, we take the ratio to $\min(p_{ref}, \sqrt{p_{ref} \times N})$, where N is the number of tasks in the DAG. There is no theoretical reason why this particular choice would be the best one. Essentially, we looked for a way to account both for the number of processors in the reference cluster and for the number of tasks. We experimented with $p_{ref} \times N$, which was too large in practice, and with $\min(p_{ref}, N)$, which was too small most of the time. Empirically, we determined that $\min(p_{ref}, \sqrt{p_{ref} \times N})$ seemed to strike a good compromise in most situations. We chose this particular function so that in a fully homogeneous system when $N \geq p_{ref}$, we end up simply using p_{ref}, which is consistent with the original CPA algorithm and thus seems natural.

Improving the second phase – One of the drawbacks of a two-phase algorithm is that the scheduling of tasks can be made more difficult due to rigid processor allocations computed in the first phase. In particular, a task may be delayed unnecessarily just because its computed processor allocation is (perhaps only slightly) larger than the number of processors available at the time when the task is ready for execution. In practice, we observed "holes" in schedules due to such a phenomenon. Consequently, we modify the second phase of HCPA as follows. Consider a task to be scheduled, whose original processor allocation was computed in phase one of the algorithm. We determine if, by using a smaller allocation, the task could be started earlier and finish no later than when using its original allocation. If so, we use the smallest such allocation.

4.2. Improved M-HEFT

A problem with M-HEFT is that it tends to use very large processor allocations for data-parallel tasks. Indeed, a task's processor allocation is chosen merely to minimizes task completion time (by contrast to the HCPA algorithm, which attempts to achieve a trade-off between T_{CP} and T_A) [9]. To remedy this problem with M-HEFT we propose three simple methods to bound a task's processor allocation:

M-HEFT-IMP – A task's allocation is increased by one processor only if that task's *execution time* is improved by more than some given threshold percentage.
M-HEFT-EFF – A task's allocation is increased by one processor only if that task's *parallel efficiency* is improved by more than some given threshold percentage.
M-HEFT-MAX – No task allocation on a cluster can be larger than some fraction of the total number of processors in that cluster.

Note that these techniques may lead to clearly suboptimal makespan in some cases. For instance, when the DAG that is a simple "chain", the best makespan is achieved by allocating all processors to each task (which is the schedule computed by the original M-HEFT algorithm). However, our goal here is to not aim solely for the best makespan, but to take into account other metrics such as the efficiency of the schedule, as seen in our experimental results.

5. Evaluation

Our goals in this section are to quantify the impact of the algorithm improvements proposed in Section 4 and to determine which of the improved HCPA or the improved M-HEFT leads to better schedules. Note that while the bulk of the results in [9] are for a comparison of HCPA and CPA, that paper also contains a succinct comparison of HCPA and M-HEFT. However, the only result from the comparison was the obvious observation that M-HEFT achieves better overall execution times than HCPA but with lower parallel efficiency, which is really due to a glaring flaw in the original M-HEFT algorithm. We have addressed this limitation in Section 4.2. Conversely, we have also improved

the HCPA algorithm. Therefore, a new comparison is necessary to truly understand which approach is more effective.

5.1. Experimental Methodology

We use simulation to explore wide ranges of application and platform scenarios in a repeatable manner and to conduct statistically significant numbers of experiments. Our simulator is implemented using the SIMGRID toolkit [6].

Simulated Platforms – We consider platforms that consist of $c = 1$, 2, 4, and 8 clusters. Each cluster contains a number of processors between 16 and 128, picked at random using a uniform probability distribution. The links connecting the processors of a cluster to that cluster's switch can be Gigabit Ethernet (bw $= 1Gb/s$ and lat. $= 100\mu sec$) or 10Gigabit Ethernet (bw $= 10Gb/s$ and lat. $= 100\mu sec$) and we simulate contention on these links. The switch in a cluster has the same bandwidth and latency characteristics at these network links, but does not experience contention. The links connecting clusters to the network backbone have a bandwidth of $1Gb/s$ and a latency of $100\mu sec$. Half the clusters use Fast Ethernet devices, and the other half use Gigabit Ethernet devices. Finally, the backbone connecting the clusters together has a bandwidth of $25Gb/s$ and a latency of $50msec$.

In our experiments we choose to keep the network characteristics fixed and we vary processor speeds to experiment with various communication/computation ratios of the platform. Processor speeds, which are measured in GFlop/sec and are homogeneous within each cluster, are sampled from a uniform probability distribution as follows. We consider a fixed number of possible minimum speeds: 0.25, 0.5, 0.75, and 1; and of heterogeneity factors: 1, 2, 5 (when there are more than one cluster in the platform). The maximum processor speed is computed as the product of a minimum speed by a heterogeneous factor. For instance, a minimum speed of 0.5 and a heterogeneity factor of 5 means that the processors have uniformly distributed speeds between 0.5 and 2.5 GFlop/sec. We assume that each processor has a 1GByte memory.

The above parameters lead to 40 platform configurations, 4 homogeneous and 36 heterogeneous (3*4*3). Since there are random components, we generate five samples for each configuration, for a total of 200 different sample platforms.

Simulated Applications – We take a simple approach to model data-parallel tasks. We assume that a task operates on a data set of n double precision elements. Since each processor has 1 GByte of memory, we assume that n can be at most $121M$. We also assume that n is above $4M$ (if n is too small, the data-parallel task should most likely be aggregated with its predecessor or successor). The volume of data communicated between two tasks is proportional to n. We model the computational complexity of a task as one of the three following forms, which are representative of many common applications: $a \cdot n$ (e.g., image processing of a $\sqrt{n} \times \sqrt{n}$ image), $a \cdot n \log n$ (e.g., sorting an array of n elements), $a \cdot n^{3/2}$ (e.g., multiplication of $\sqrt{n} \times \sqrt{n}$ matrices), where a is picked randomly between 2^6 and 2^9. As a result different tasks exhibit different communication/computation ratios. We consider four scenarios: three in which all tasks have one of the three computational complexities above, and one in which task computational complexities are chosen randomly among the three. Finally, we assume that a fraction α of a task's sequential execution time is non-parallelizable, with α uniformly picked between 0% and 25%.

We consider applications that consist of 10, 20, or 50 data-parallel tasks. We use four popular parameters to define the shape of the DAG: width, regularity, density, and "jumps". The width determines the maximum parallelism in the DAG, that is the number of tasks in the largest level. A small value leads to "chain" graphs and a large value leads to "fork-join" graphs. The regularity denotes the uniformity of the number of tasks in each level. A low value means that levels contain very dissimilar numbers of tasks, while a high value means that all levels contain similar numbers of tasks. The density denotes the number of edges between two levels of the DAG, with a low value leading to few edges and a large value leading to many edges. These three parameters take values between 0 and 1. In our experiments we use values 0.1, 0.2, and 0.8 for width and 0.2 and 0.8 for regularity and density. Finally, we add random "jumps edges" that go from level l to level $l + jump$, for $jump = 1, 2, 4$ (the case $jump = 1$ corresponds to no jumping "over" any level). We refer the reader to our DAG generation program and its documentation for more details [4]. Our DAG generation procedure is similar to ones used previously (e.g., in [5]).

Overall, we have $4^2 \times 3^3 = 432$ different DAG types. Since some elements are random, for each DAG type we generate three sample DAGs, for a total of $1,296$ DAGs.

Simulation Procedure Each experiment consists in simulating one application configuration on one platform configuration, for a total of $200 \times 1,296 = 259,200$ scenarios. For each scenario, we compare the original and modified versions of HCPA and M-HEFT, using the following two classic metrics.

Our first metric is *makespan*, that is the time elapsed between the beginning of the first application task and the completion of the last application task (in seconds). Our second metric is *efficiency*. The traditional definition of parallel efficiency is difficult to generalize on a heterogeneous platform. Instead, we define efficiency as the ratio of the total work in the sequential application execution on the fastest processor in the platform to the total work of the parallel execution. The *total work* is defined as the sum of

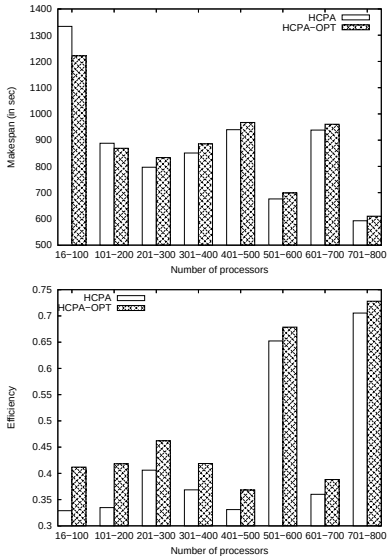

Figure 1. Average makespan and efficiency vs. the total number of processors in the computing platform, for the original HCPA algorithm and the algorithm modified with a new stopping criterion for the allocation phase (HCPA-OPT).

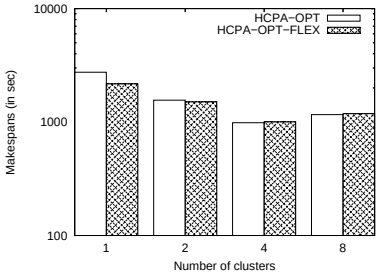

Figure 2. Average makespan versus the number of clusters in the platform, for the HCPA algorithm with a modified stopping criteria for the allocation phase (HCPA-OPT) and the algorithm further modified by allowing task allocations to be changed during the scheduling phase (HCPA-OPT-FLEX).

the work of each task in the application, that is the product of a task's execution time and of the number of processors it used, scaled by the processor speed. A high efficiency value indicates that compute resources were used effectively to improve the makespan, which is important for budget-minded users when resources are not free (the user gets more "bang for the buck"). In such a case, a longer makespan for a higher efficiency may be a desirable tradeoff. Note that this efficiency metric does not account for "holes" in the schedule, which are bound to occur when using list scheduling heuristics in 2-phase scheduling algorithms for instance. Our rationale is that the schedule should be implemented so that the user is not "charged" for unused processors. In current systems, which are controlled by batch schedulers, this can be achieved via multiple batch submissions and/or reservations. The brute force approach that consists in reserving as many processors as the maximum number of processors needed throughout the application execution for the whole duration of that execution is obviously very wasteful for DAG-structured applications.

5.2. Results

Improved HCPA – Fig. 1 shows the performance of the original HCPA algorithm and that of the algorithm with the modified $T_{CP} < T_A$ stopping criterion described in Section 4.1 (denoted by HCPA-OPT in the figure), for our two metrics versus the number of processors in the platform. We can see that although the modification does not increase makespan significantly it can lead to substantially higher efficiency. More specifically, the average makespans achieved with the HCPA-OPT algorithms are at most 4.6% worse than those achieved by HPCA (and up to 8% better for small numbers of processors). Efficiency is always better with HCAP-OPT, from 3% better for large numbers of processors to 25% better for small numbers of processors.

As mentioned in Section 4.1, our modification to HPCA was intended to deal with the cases in which the number of processors is larger than the number of tasks in the application. In our experiments, the largest number of tasks in an application is between 10 and 50. Therefore, even for numbers of processors as low as 16 we can see improvements for many or in fact all applications. We were able to determine that these improvements were indeed due to the modified stopping criterion.

One may then wonder why the improvements are only marginal for numbers of processors larger than 500. In our simulations these larger numbers of processors are mostly for platforms with 8 highly heterogeneous clusters. In these situation, both algorithms tend to restrict themselves to using only the faster clusters. Therefore, large allocations on the reference clusters often cannot be translated into feasible allocations, as explained in Section 3.1. In these cases, the allocation phase does not stop due to the (modified or unmodified) $T_{CP} < T_A$ stopping criterion at all. Instead, the allocation phases stops when task allocations on the reference cluster cannot translated into feasible allocations. Consequently, HCPA and HCPA-OPT, both using the same stopping criterion, compute the same schedule.

Fig. 2 shows the improvement in makespan due to the second modification described in Section 4.1 (denoted by HCPA-OPT-FLEX in the figure), versus the number of clusters in the platform. This figure is only for cases in which there was an opportunity for using the modification, which accounted for only about 4% of all cases in our experiments. We can see that the improvement is significant only when the number of cluster is small. We conclude that our proposed modification is most likely not useful for large platforms that consist of many heterogeneous clusters. Interestingly, the improvement is the most frequent (for above 7% of the cases) and the largest (above 15%) for single-cluster platforms. Therefore our proposed modification would be a good addition to the original CPA algorithm.

Algorithm	Makespan	Efficiency
M-HEFT	815.52	0.105
M-HEFT-IMP5	+92.3%	+487.6%
M-HEFT-EFF50	+23.4%	+308.1%
M-HEFT-MAX50	-6.2%	+71.6%

Table 1. Average makespan and efficiency for the original M-HEFT algorithm over all experiments, and the impact of our three proposed modifications to M-HEFT.

Improved M-HEFT – Table 1 shows the impact of the three modifications to the M-HEFT algorithm proposed in Section 4.2. M-HEFT-IMP5 denotes the M-HEFT algorithm modified so that a processor is added to a task's allocation only if that addition leads to an improvement in the task's execution time larger than 5%. M-HEFT-EFF50 denotes the M-HEFT algorithm modified so that a task's allocation is never so large that the task's parallel efficiency is under 50%. Finally, M-HEFT-MAX50 denotes the M-HEFT algorithm modified so that a task's allocation cannot use more than 50% of the processors of a cluster.

Based on the results in the table, one can see several trade-offs. For instance, using M-HEFT-IMP5 almost doubles the makespan but improves efficiency by roughly a factor 6, thereby saving a significant amount of compute resources. Only the simplest modification, i.e., M-HEFT-MAX50, leads to results strictly superior to that achieved by M-HEFT. On average, M-HEFT-MAX50 reduces the makespan (by avoiding large allocations that could have a negative impact on the length of the critical path) but also improves efficiency (also by avoiding large allocations). However, due to their large improvement in efficiency, M-HEFT-EFF50 and M-HEFT-IMP5 may be better choices for very budget-minder users.

We have experimented with different values for the threshold values used in our modifications to M-HEFT. As expected, for all three modifications a variety of trade-offs between makespan and efficiency can be achieved by tuning the threshold values. How to do this tuning is rather subjective as the "best" trade-off depends on the user's goals (e.g., more or less budget-minded, more or less makespan-driven). Nevertheless we found two interesting facts. First, using values higher than 5% for M-HEFT-IMP shows steeply diminishing returns and should most likely be avoided. Second, the M-HEFT-MAX modification becomes better relative to the other two modifications as the application DAG becomes larger and wider. This is because M-HEFT-MAX enforces that "free space" be left on all clusters, which can be used effectively to execute large numbers of concurrent tasks. The particular threshold values we picked for the results shown in Table 1, i.e., 5%, 50%, and 50%, although arbitrary are reasonable (i.e., neither leading to the lowest or the highest makespan, or efficiency, when compared to results for the range of possible values).

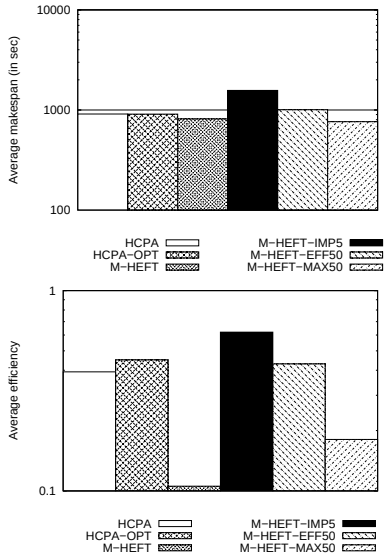

Figure 3. Average makespan and efficiency for original and modified algorithms.

Comparing the two approaches – Fig. 3 shows results, averaged over all $259,200$ experiments, for the original and modified algorithms. We find that both the original HCPA and M-HEFT algorithms show results strictly inferior to one or more of their modified versions, which confirms results discussed in the previous section. No algorithm is best, which we expected since we use both makespan and efficiency as metrics. (Note that all algorithms improve the sequential makespan, not shown on the figure, by at least a factor 7). The comparison between the HCPA-OPT and the three M-HEFT versions is as follows:

- HCPA-OPT is strictly superior to M-HEFT-EFF50.
- On average HCPA-OPT's makespan is 20% larger than that of M-HEFT-MAX50, but its efficiency is 150% higher.
- On average HCPA-OPT's makespan is 42% smaller than that of M-HEFT-IMP5, but its efficiency is 27% lower.

Although the choice of the algorithm depends on the particular goals of individual users, we can draw broad conclusions. The M-HEFT-MAX50 algorithm is a bit of an extreme (while not as extreme as the original M-HEFT) and is most likely of interest only to users that are not budget constrained. The two remaining contenders are HCPA-OPT and M-HEFT-IMP5. The question that a user should ask is: "Is a 42% improvement in makespan worth a 27% loss in efficiency?" HPCA-OPT should be used if the answer to this question is "Yes", and M-HEFT-IMP5 otherwise. Our personal guess is that most users would answer Yes to this question and opt for using HPCA-OPT.

6. Conclusion

In this paper we have studied the scheduling of a mixed parallel application onto a multi-cluster heterogeneous computing platform. We have improved on the two scheduling approaches that have been proposed in the literature: (i) Approach #1 adapts two-phase scheduling algorithms for mixed-parallel applications on homogeneous platforms to make them amenable to heterogeneous platforms; (ii) Approach #2 adapts list scheduling algorithms for task-parallel applications on heterogeneous platforms to make them amenable to mixed parallelism. We found that no algorithm emerges as a clear winner. However, it seems that Approach #1 leads to the most likely desired trade-off between makespan and efficiency. One advantage of Approach #2 is that the algorithms are tunable with simple parameters to achieve trade-offs between performance and efficiency. Our conclusion is that Approach #1 is appropriate for the vast majority of the users, but that sophisticated users could opt from Approach #2.

A promising future direction for this work is to consider an alternative model for the performance of data-parallel tasks, especially one that captures the performance impact of intra-task communications when different clusters may use different interconnect technologies (i.e., slower or faster switches). We are also planning to run experiments on real-world platforms to confirm the results of our simulations.

Acknowledgments

The authors wish to thank the reviewers for their insightful comments. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see `https://www.grid5000.fr`).

References

[1] V. Boudet, F. Desprez, and F. Suter. One-Step Algorithm for Mixed Data and Task Parallel Scheduling Without Data Replication. In *17th Int. Parallel and Distributed Processing Symposium (IPDPS)*, Apr. 2003.

[2] H. Casanova, F. Desprez, and F. Suter. From Heterogeneous Task Scheduling to Heterogeneous Mixed Parallel Scheduling. In *10th Int. Euro-Par Conference*, volume 3149 of *LNCS*, pages 230–237, Aug. 2004.

[3] S. Chakrabarti, J. Demmel, and K. Yelick. Modeling the Benefits of Mixed Data and Task Parallelism. In *Symp. on Parallel Algorithms and Architectures*, pages 74–83, 1995.

[4] DAG Generation Program. `http://www.loria.fr/~suter/dags.html`.

[5] Y.-K. Kwok and I. Ahmad. Benchmarking and Comparison of the Task Graph Scheduling Algorithms. *JPDC*, 59(3):381–422, 1999.

[6] A. Legrand, L. Marchal, and H. Casanova. Scheduling Distributed Applications: The SimGrid Simulation Framework. In *3rd IEEE Symp. on Cluster Computing and the Grid (CCGrid)*, pages 138–145, May 2003.

[7] R. Lepère, D. Trystram, and G. Woeginger. Approximation Algorithms for Scheduling Malleable Tasks Under Precedence Constraints. *Int. Journal on Foundations of Computer Science*, 13(4):613–627, 2002.

[8] W. Ludwig and P. Tiwari. Scheduling Malleable and Nonmalleable Tasks. In *Symp. on Discrete Algorithms (SODA)*, pages 167–176, 1994.

[9] T. N'Takpé and F. Suter. Critical Path and Area Based Scheduling of Parallel Task Graphs on Heterogeneous Platforms. In *12th Int. Conf. on Parallel and Distributed Systems (ICPADS)*, pages 3–10, July 2006.

[10] A. Radulescu and A. van Gemund. A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling. In *15th Int. Conf. on Parallel Processing (ICPP)*, Valencia, Spain, Sept. 2001.

[11] S. Ramaswany. *Simultaneous Exploitation of Task and Data Parallelism in Regular Scientific Applications*. PhD thesis, Univ. of Illinois, Urbana-Champaign, 1996.

[12] T. Rauber and G. Rünger. Compiler Support for Task Scheduling in Hierarchical Execution Models. *Journal of Systems Architecture*, 45:483–503, 1998.

[13] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE TPDS*, 13(3):260–274, 2002.

[14] J. Turek, J. Wolf, and P. Yu. Approximate Algorithms for Scheduling Parallelizable Tasks. In *Symp. on Parallel Algorithms and Architectures*, pages 323–332, 1992.